Hierarchical reconfiguration of FPGAs

DOI:
10.1109/FPL.2014.6927491

Hierarchical Reconfiguration of FPGAs

Dirk Koch
The University of Manchester
Email: dirk.koch@manchester.ac.uk

Christian Beckhoff
University of Oslo
Email: christian@ReCoBus.de

***Abstract*—Partial reconfiguration allows some applications to substantially save FPGA area by time sharing resources among multiple modules. In this paper, we push this approach further by introducing hierarchical reconfiguration where reconfigurable modules can have reconfigurable submodules. This is useful for complex systems where many modules have common parts or where modules can share components. For such systems, we show that the number of bitstreams and the bitstream storage requirements can be scaled down from a multiplicative to an additive behavior with respect to the number of modules and submodules. A case study consisting of different reconfigurable softcore CPUs and hierarchically reconfigurable custom instruction set extensions demonstrates a $18.7\times$ lower bitstream storage requirement and up to $10\times$ faster reconfiguration speed when using hierarchical reconfiguration instead of using conventional single-level module-based reconfiguration.**

I. Introduction

Partial reconfiguration is a technique for sharing FPGA resources among multiple modules that are executed mutually exclusively to each other. By loading only the entire currently needed modules to an FPGA, a smaller and consequently cheaper and less power hungry device can be used. Examples for such systems have been presented several times before.

For instance, in [1], [2], [3], partial reconfiguration was used for adapting systems to environmental changes (e.g., light conditions in a vision system or channel quality in a wireless communication system). As the environment can only be in one specific state, specialized hardware accelerators were used for adapting a system at run-time.

A more flexible system using partial reconfiguration was proposed by Dennl et al. in [4]. In that system, reconfiguration is used for accelerating database processing by composing a chain of SQL operator modules together at run-time for executing SQL queries. This system uses module relocation and multi-instantiation of modules which is not available when using the Xilinx vendor tools.

A. Hierarchical Reconfiguration

Common for all these examples is that modules are placed into a static system that provides the communication infrastructure for hosting the reconfigurable modules. In this paper, we introduce a methodology for implementing reconfigurable modules hierarchically inside reconfigurable modules. In this case, a reconfigurable module will provide the communication infrastructure for hosting reconfigurable modules and not a surrounding static system. However, the static system will still be in charge for managing the configurations and for sending configuration data to the FPGA.

Hierarchical reconfiguration allows the efficient implementation of many modules that have common parts or that can share components. For example, consider a sorter accelerator for the reconfigurable database accelerator, presented in [4]. Then the sorter should only be configured to the device if needed to process a given query. However, for sorting different data types, including integer, floating-point, and text, different sorters have to be implemented and provided by the system. By using hierarchical reconfiguration, someone can consider a system that consists of a universal sorter providing the data movement and storage functionality which is able of embedding reconfigurable submodules that perform the data type specific operations (e.g., integer, floating-point, or string compare).

B. Hierarchical Multicore System

As a case study, we will examine how reconfigurable custom instruction set extensions can be used by different reconfigurable softcore CPUs in a dynamic multicore system. The concept is illustrated in Figure 1. The system allows adapting to different workload scenarios by loading specialized instances of softcore CPUs or dedicated hardware accelerators into a reconfigurable region. Assuming that the static system is the level-0 configuration, these modules represent level-1 configurations. Some level-1 modules can themselves hierarchically host reconfigurable modules which are level-2 modules. We use the term reconfigurable submodule and level-2 module interchangeable in this paper to distinguish from reconfigurable level-1 modules. Note that more than two levels of partial reconfiguration might be used for large FPGAs. For example, we could consider our case study with a static system (level 0) hosting a reconfigurable CPU (level 1) that can be dynamically extended with a crypto accelerator (level 2) that itself provides the secret key as a level 3 sub-subodule.

In the example in Figure 1, it should be noticed that all partial modules, regardless if they are level-1 or level-2, are relocatable to different positions. However, this might follow restrictions due to resource constraints such as different resources for logic and memory on the FPGA fabric.

C. Advantages

By providing multiple level-1 modules in a bitstream library at run-time, more specialized modules can be used in a system allowing for better performance at lower FPGA cost. For instance, a reconfigurable softcore CPU might only provide the required instructions to run a specific task while

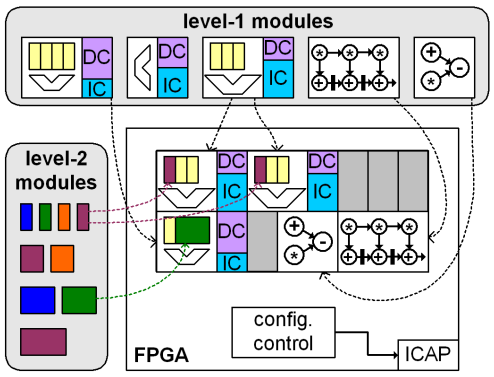

Figure 1. Hierarchical reconfiguration. Some level-1 modules can host level-2 modules. Note that the level-1 softcore CPUs vary with respect to data cache (DC), instruction cache (IC), and the number of resources available to host level-2 modules.

removing all logic for instructions that will never be used by this task. This concept is known as ISA subsetting [5]. In addition, there might be different versions of the softcore CPU with different instruction cache and data cache sizes for better matching memory requirements of different tasks. Another adaptation which cannot be efficiently carried out by the user logic of most FPGAs, but by reconfiguration, is changing to subword multiplication (e.g., changing a 32-bit multiplier to two 16-bit or four 8-bit multipliers). In our case study (see Section IV), we will use a very generic MIPS softcore CPU specification that can be parameterized with different memory layouts and that supports ISA subsetting for generating level-1 softcore CPUs based on profiling information [6].

In addition to the level-1 modules, reconfigurable instruction set extensions (implemented as reconfigurable level-2 submodules) allow a second way of customizing some of the softcore CPUs. For example, a level-1 CPU might use a reconfigurable custom instruction set extension for firstly computing a SHA256 checksum which is replaced after completion by a reconfigurable AES-round. So instead of providing two level-1 configurations, one level-1 and two small level-2 configurations are sufficient to be stored in the partial module bitstream repository of the assumed system. If we further assume that both custom instructions require 25% of the logic resources of a customized level-1 CPU, then the variant with two level-2 instructions can save up to $(100\% - 25\%)/2 = 37.5\%$ of the memory requirements in the module repository ($2 \times 100\%$ versus $75\% + 2 \times 25\%$). This effect is visualized in Figure 2a) and Figure 2b).

More important than the configuration memory savings is that swapping between different modules can be carried out much faster, because fewer resources have to be reconfigured when only reconfiguring the level-2 submodules.

At first glance, this approach seems to be more difficult to implement as more different modules might have to be provided. For example, the module repository in Figure 2b) needs less memory but has to store three instead of two modules, when using hierarchical reconfiguration. However, when scaling the approach up to more level-1 and level-2 modules, less modules will be needed and the memory requirements get even stronger reduced.

Let M be the set of level-1 modules and N be the set of level-2 submodules, let further $f(m,n)$ be a characteristic function that returns 1 if the system uses a specific combination of a level-1 module (m) with a level-2 module (n) and that returns 0 else, then traditional single level reconfiguration needs a repository with

$$r_{level1} = \forall\ m \in M,\ n \in N\ \sum f(m,n) \qquad (1)$$

modules. The number of modules for hierarchical reconfiguration is:

$$r_{level2} = |M| + |N| \qquad (2)$$

If all permutations of level-1 and level-2 modules can occur in a system, Equation 1 becomes $r_{level1} = |M| \cdot |N|$. In other words, the total number of modules scales multiplicative for the traditional single-level reconfiguration, but only additive when using hierarchical reconfiguration. Consequently, for large systems with a large variety of different modules, hierarchical reconfiguration results in a much cleaner design flow. Note that the improvement from multiplicative to additive complexity still holds if we implement design alternatives to incorporate the heterogeneous resource layout (e.g. the position of logic, multiplier and memory resources) of most commercial FPGAs.

We can also say that hierarchical reconfiguration adds extra programmability to a reconfigurable system. In the software world, applications are commonly built from libraries that again may be built from other libraries and so forth. Hence, various applications can re-use library functions in a layered hierarchical manner. For FPGAs, hierarchical reconfiguration allows adapting this concept to the hardware world. In both cases, pre-implemented functions (object code and fully routed netlists respectively) are plugged together using some kind of interface definition. Consequently, hierarchical reconfiguration can pave a way for easier IP reuse and for higher acceptance of FPGAs by software developers.

D. Paper Organization

The paper continues with a discussion about the design flow for implementing hierarchical systems in the next section. This flow is based on the tool GOAHEAD that will be compared with other partial reconfiguration design tools in Section III. After this, we present a case study using hierarchical reconfiguration for a multicore system in Section IV.

II. Design Flow

The following sections describe the design flow for implementing hierarchical reconfigurable systems. The flow shares techniques known from conventional module-based partial reconfiguration and focus is put on the parts that are specific for allowing hierarchical reconfiguration. This work is based on the GOAHEAD tool [7] in which the static system and the partial modules are implemented independently from each other. Consequently, we will describe the implementation of the static system (level-0), the partial modules (level-1), and the partial submodules (level-2) in different paragraphs.

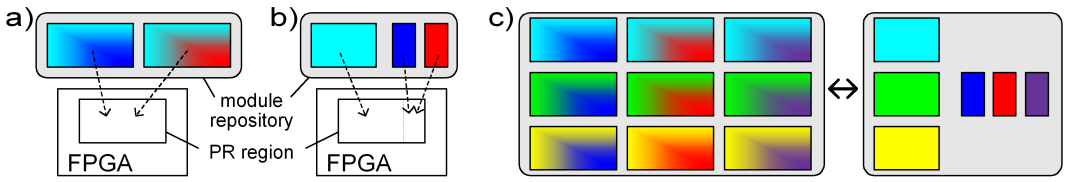

Figure 2. a) Conventional single level reconfiguration where most of the resources are identical between the two modules, b) Saving repository space by using hierarchical reconfiguration. Note that this allows also for faster reconfiguration, if only the submodules will be swapped. c) Visualization of the savings in the module repository if a system uses three level-1 and three level-2 modules. Note that not only the repository requirements are reduced, but that also the total number of partial modules is less when using hierarchical partial reconfiguration.

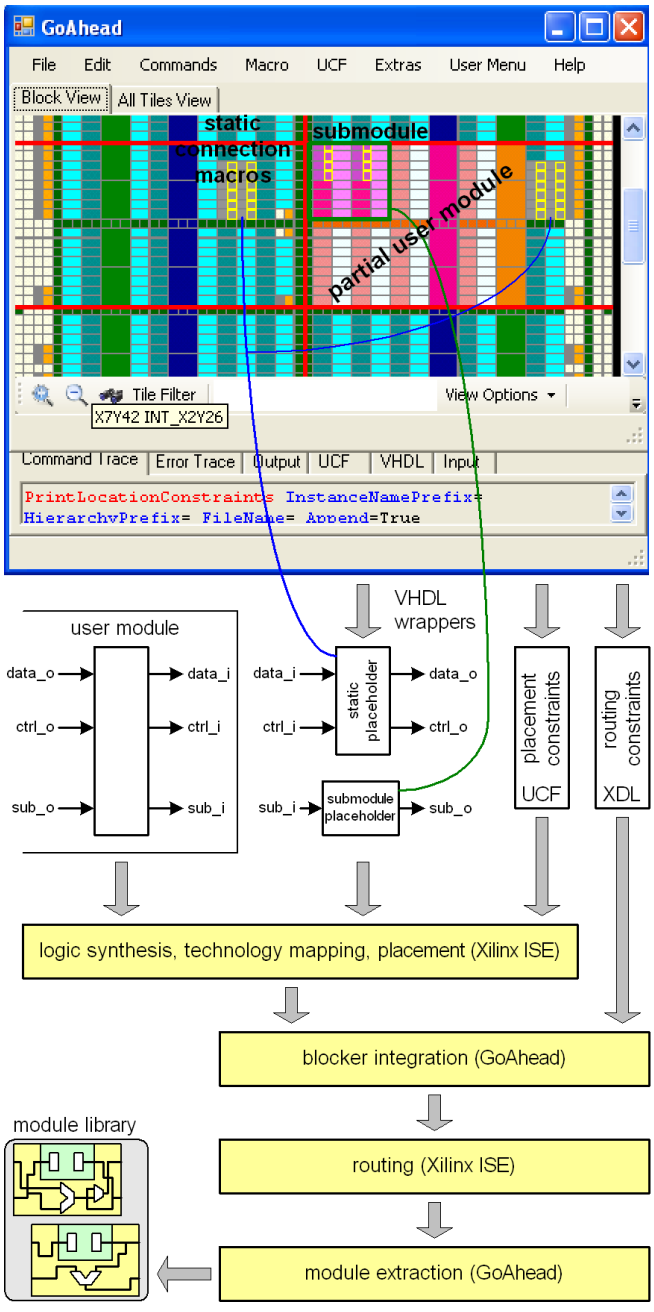

Figure 3. Partial module (level-1) design-flow.

A. Level-0: Static System

There are no special requirements for implementing the static system of a hierarchically reconfigurable system as compared to a non-hierarchical system. However, the reconfiguration of the FPGA might need to update parts in the user logic of the module that is loaded to the fabric. In this case, the static system has to provide a communication link for sending corresponding data to the module. For this work, we used the Atlys Spartan-6 demonstration system which is distributed with GOAHEAD [7]. This system provides fine-grained two dimensional module placement including a communication architecture tailored to streaming applications. The design flow of the static system is illustrated in Figure 5.

B. Level-1: Partial Modules

Each level-1 module is implemented individually and independently to the static system or any other module or submodule. Consequently, there is no rule in which order the different parts of the system will be implemented. Therefore, a bottom-up as well as a top-down design methodology can be used for hierarchical reconfiguration.

1) Floorplanning: A partial module is implemented by constraining its resources into a bounding box. GOAHEAD supports this Floorplanning with a GUI as shown in Figure 3. Inside the area of the partial user module, a subregion can be defined, if the partial module should be able of hosting submodules. Note that submodules are optional in level-1 partial modules.

For floorplanning, it is required that the bounding box of a partial module provides sufficient resources for hosting the user logic of the partial module as well as the resources needed for hosting the submodules. For budgeting the resource requirements, we perform an initial synthesis run for partial modules and submodules, if resource requirements cannot be determined beforehand.

After this, connection macros have to be placed. These macros are LUTs that act as sink and source primitives for connecting the top-level interface signals of a user module with wires outside the partial module bounding box. This means that the static system is substituted with static system connection macros and, in the case that partial submodules will be used, that the level-2 submodule is substituted by submodule connection macros. The submodule connection macros have to be placed inside the submodule region.

2) Constraints Generation, Synthesis, and Placement: From the floorplanning information, we used GOAHEAD for

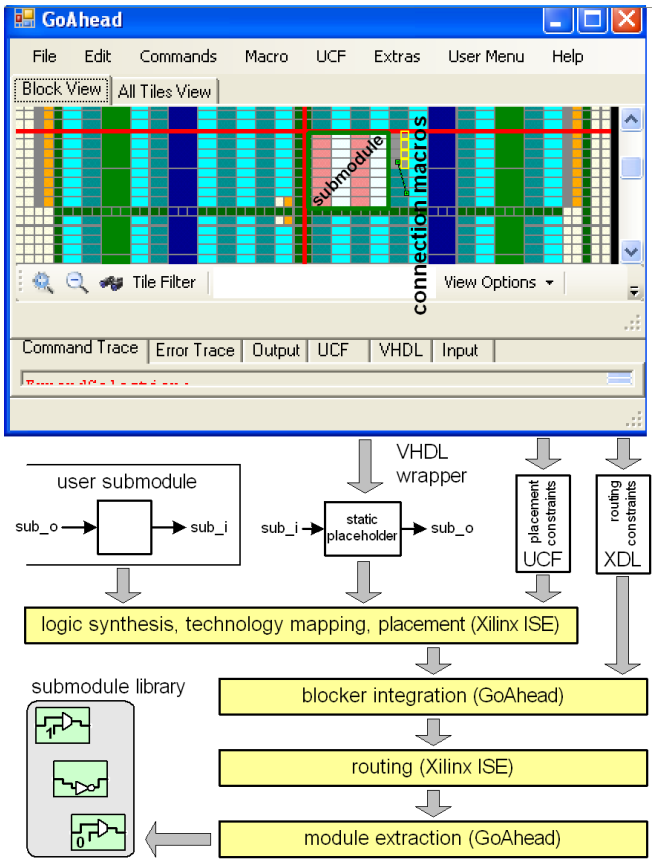

Figure 4. Partial module (level-2) design-flow.

creating VHDL-wrappers which instantiate the connections macros (which are described in the previous paragraph). These wrappers act as substituting placeholder modules for the static system and the level-2 submodules. The wrappers have to be connected directly with the user module and GOAHEAD creates all VHDL code for this step.

In addition to the placeholder wrapper modules, placement constraints will be generated for the connection macro placement and further constraints forcing the placement of all primitives of the user module into the partial module bounding box. These constraints are used together with optional other user-defined constraints to run the Xilinx vendor tools for synthesis, technology mapping and placement.

3) Routing and Module Extraction: In addition to the location constraints, we used the GOAHEAD blocker generation feature to create two blocker macros. The blockers occupy all routing resources in a predefined region. By this, we prohibit the blocked wires to be used in following routing steps. These blockers are added into the placed design before running the router. One blocker pre-occupies all routing resources outside the partial module bounding box and the second blocker (which is only needed when using level-2 reconfiguration) pre-occupies all routing resources inside the region allocated for submodules. However, wires that will be used to connect the partial top-level interface signals with the connection macros (used to substitute the static system and the submodules) will be removed from blocking. This results in tunnels inside the blockers leaving dedicated routing paths to the connection macros. This process is shown in Figure 5b).

The connection macros are used to connect all top-level interface signals except for clock signals. For providing one or more clocks to the submodules, clock signals are routed to all blocker primitives inside the level-2 submodule area. This will speculatively enable all clock tree drivers in this area (as proposed in Xilinx application note XAPP290 V1.1).

After the blocker insertion, we run the Xilinx production line router (par). Next, we remove the blocker routing from the fully routed netlist and use the original floorplan information to cut out the partial module. This process is repeated for all partial level-1 modules. The blocker macros and placement constraints can be reused, if multiple modules share the same bounding box (i.e., resources). With these steps, we are building up a module library consisting of fully placed and routed level-1 modules. Note that after floorplanning all steps are carried out fully automatic and manual interaction is only needed if resource requirements demand a resizing of bounding boxes.

C. Level-2: Partial Submodules

The design flow for partial level-2 submodules is very similar to the level-1 module design flow except that we are not reserving resources for a submodule. Figure 4 shows the floorplan and design-flow for a partial submodule that fits into the partial module floorplan which is shown in Figure 3.

Note that the level-1 module is designed with two adjacent regions for hosting submodules. This is achieved by providing individual connections in both regions (see the connection macro placement inside the submodule region in Figure 3. This can be used to place either up to two small submodules or one larger submodule. The example in Figure 4 shows the floorplan for a large submodule. The flexibility provided in this system reduces substantially internal fragmentation overhead.

D. Static Route-Throughs

Reserving a larger region for hosting reconfigurable modules can result in a very congested design with poor performance, if signals have to route around that region. GOAHEAD solves this issue by allowing static signals to cross a reconfigurable region. The described wire blocking for constraining the routing basically implements an exclusive allocation of routing resources to the static system, a reconfigurable module, or a submodule. For implementing static route-through signals, local long distance wires are allocated to the static system such that these wires form a straight predefined routing path through a reconfigurable region. All these routing paths will then be programmed speculatively in the level-1 modules and level-2 submodules regardless if a particular path is used or not by the static system. When allocating static paths regular structured (e.g., by allocating the same equivalent wire in all switch matrices within the horizontal or vertical span of a reconfigurable region), route-throughs can be used in hierarchically reconfigurable

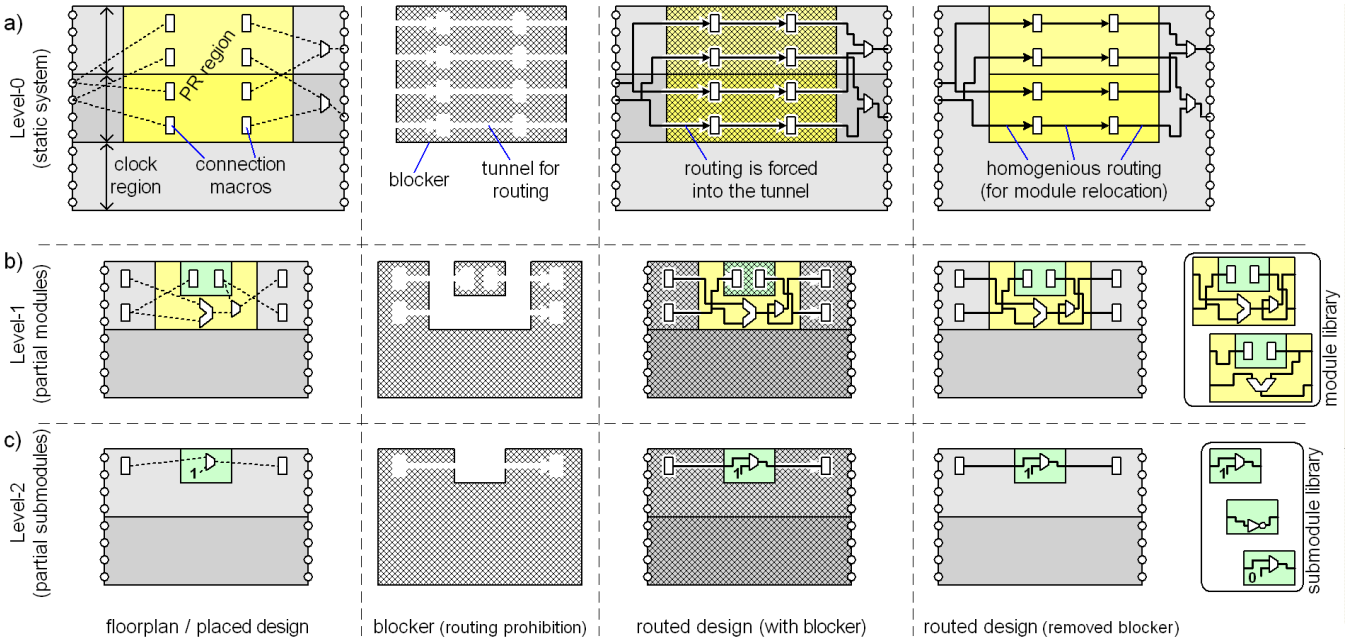

Figure 5. Phases at the different design levels. a) The static system follows the default GOAHEAD flow with connection macros used to substitute the reconfigurable modules and blockers which occupy all routing resources except for an optional tunnel. The blocker is created by GOAHEAD and temporarily embedded into the design only for routing the design. b) Partial modules are implemented by substituting the static system with connection macros. In addition, connection macros are used within the hierarchical PR region to define the routing with the submodules (together with a blocker). These phases have to be carried out for each level-1 module and the final physically implemented module netlists can be extracted from the design and stored in a module library. c) Submodules are implemented individually following the default GOAHEAD flow. The result is a submodule library.

systems together with module relocation. The corresponding constraints generation is supported in GOAHEAD.

E. Bitstream Generation

All bitstreams are generated with the module placer available in GOAHEAD and the Xilinx bitstream generation tool (bitgen), as described in the following sections.

1) Level-0: Start-up Bitstream: For generating the start-up configuration, we can run the usual vendor bitstream generation. Alternatively, we can create a start-up bitstream that already contains certain reconfigurable modules. In this case, we load the (fully placed and routed) static level-0 design into GOAHEAD. After this, all initial level-1 modules will be placed, followed by all initial level-2 submodules. Note that the original connection macros will be overwritten during this step. Consequently, the connection macros are not resulting in any logic overhead and reconfigurable regions can be entirely used for implementing partial modules. The result of this netlist merging is a complete netlist that can be used with the Xilinx bitstream generation tool for generating a full initial configuration bitstream containing already some reconfigurable modules.

The resulting netlist can also be directly used for static timing analysis. We use this feature for checking the initial system timing, but also the timing of relocateable modules at critical positions. Critical positions are, for example, positions where the routing to other parts of the system has to take long distances.[1] Another critical placement position is the middle column of the FPGA fabric that includes the configuration logic. While the configuration logic is transparent to the user, it still takes area on the die, hence, resulting in longer routing paths for wires spanning over that logic [8]. By relocating modules to different positions on this middle column might consequently result in timing violations which can be analyzed at design-time.

2) Level-1: Partial Module Bitstreams: Partial module bitstreams are generated by placing each module one after the other to each possible placement position into the static system on a netlist level with the help of GOAHEAD. Then, a differential bitstream is generated for each resulting design using the Xilinx vendor bitstream generation tool (bitgen -r). This bitstream contains the configuration data needed to reconfigure from a static system to a static system that is including the partial module at a specific placement position.

With this process, we verify that the partial bitstreams that are generated by the run-time system reassemble bitstreams that the vendor tools would generate. At run-time, only address fields inside the bitstreams have to be adjusted according to the module placement position. This process is well documented in the device-specific user guides.

3) Level-2: Partial Submodule Bitstreams: We can now adapt the methodology from the previous paragraph for generating the level-2 partial submodule bitstreams. Let the

[1]In general, it is a good design practice to include registers for all input and output signals of parts in the system that are involved in the communication with partial modules. This includes the static system as well as the modules and submodules). Then (in most cases) only the internal timing of modules is needed to be verified.

input be one level-0 netlist (static), a set of level-1 modules (L1_modules) with corresponding placement positions (L1_positions), a set of level-2 submodules (L2_modules) with relative placement positions inside the level-1 modules (L2_positions). Let us further define a function *MergeNetlists*(start, increment, position) that adds the netlist increment into the netlist start placed at position. Then we can describe the partial bitstream generation for submodules using the following pseudo code.

```
1: forall (m_1 ∈ L1_modules) do
   {
2:   forall (p_1 ∈ L1_positions(m_1)) do
     {
3:       L1_netlist = MergeNetlists(static, m_1, p_1)
   \\ generate level-1 module bitstreams:
4:       bitfile_m_1_p_1 =
                GenDiffBitstream(static, L1_netlist)
5:       forall (m_2 ∈ L2_modules) do
         {
6:         forall (p_2 ∈ L2_positions(m_2)) do
           {
7:             L2_netlist =
               MergeNetlists(L1_netlist, m_2, (p_1 + p_2))
   \\ generate level-2 submodule bitstreams:
8:             bitfile_m_1_p_1_m_2_p_2 =
               GenDiffBitstream(L1_netlist, L2_netlist)
    }  }  }  }
```

Line 4 was added to show the generation of the partial level-1 module bitstreams, as described in the last paragraph. As can be seen, we iterate over a four times nested loop consisting of all level-1 modules and placement positions (lines 1–2) as well as over all submodules and corresponding relative placement positions (lines 5–6). If we consider a rather large system with 20 modules, each with 10 placement positions and 5 possible submodules, again each with 2 relative placement positions, then $20 \times 10 = 200$ level-1 module bitstreams have to be generated and $20 \times 10 \times 5 \times 2 = 2000$ level-2 submodules. If we assume about half a minute of processing time per bitstream (note that we only stitch netlists together without running place&route) the whole bitstream generation process will roughly take a full day. Luckily, the problem is fully data parallel for each module and could consequently be easily distributed to a compute farm.

The last discussion shows a limited scalability for the proposed bitstream generation which is still feasible for practical systems we can imagine today but not for large scale systems in the future. For scaling this approach up, more information about the bitstream format is needed (which can be derived from the here presented bitstream generation algorithm). Then only $20 + 5 = 25$ bitstreams are needed in total, if modules can be arbitrary relocated. However, as mentioned before, the here proposed methodology ensures valid configurations without relying on non-disclosed information.

In general it is not recommended to exchange one reconfigurable module by directly overwriting it with another module configuration because this causes transient short circuits [9]. Consequently, we generate a blanking bitstream together with each level-1 or level-2 module bitstream such that we can remove a module in a reconfigurable area before sending a new module to the FPGA.

F. Design Automation

The last paragraphs revealed that implementing hierarchical reconfigurable systems appears to be more complex than traditional static or reconfigurable systems. However, many of the described steps are well supported by GOAHEAD and need only little manual interaction.

The partitioning of the system in the different configuration levels has to be done manual, as common for all reconfigurable design flows. In general, it is not possible to automatically decide if parts or modules in a system are executed mutually exclusive such that they can time-share a reconfigurable region on an FPGA. For example, because of the halting problem, we cannot know if a task will terminate and freeing up its resources, if we assume a Turing complete model of execution. However, a designer is typically aware about the modules that are mutually exclusive to each other (which are reconfigurable level-1 candidates) or if functional blocks can be reused by different modules (which are level-2 submodule candidates).

The floorplanning of the static system (level-0), non-hierarchical level-1 modules and the level-2 submodules can be carried out automatically as described detailed in [10].

III. COMPARISON WITH OTHER PR TOOLS

This paper is based on the GOAHEAD design flow which differs from the Xilinx vendor PR flow [11]. While GOAHEAD follows a strict encapsulation of the static system and all modules, the Xilinx flow is based on an incremental design methodology. Using the vendor flow, a static system is built by leaving reconfigurable regions unused; and similar to the GOAHEAD flow, connection macros (called proxy logic by Xilinx) are placed into the PR region for implementing the communication with the partial modules. However, the routing to the connection macros is not further constrained and is in general different for each region. Then for each partial module, an incremental design, including additional placement of module primitives and incremental routing is performed followed by the generation of a partial differential bitstream. As the static routing is different in each reconfigurable region, modules cannot be relocated. The same would apply when trying to use the Xilinx design flow in hierarchical reconfigurable systems. Then the routing is in general different for each combination of a level-1 module and a level-2 submodule.

Considering the example from Section II-E3 with 20 modules, 10 partial regions, 5 possible submodules, and 2 level-2 regions for hosting submodules, 2000 incremental place&route and bitstream generation steps are needed resulting in 2000 partial module bitfiles to be provided by the run-time system. Even worse, a single change in the static system would force us to repeat these steps as the routing to the proxy logic will change in general. In contrast, GOAHEAD allows individual modifications to the static

system without any interference with the partial modules or submodules.

With OpenPR [12] there exists another partial reconfiguration tool which is targeting a more scalable design flow than available by Xilinx. However, that tool supports less devices and uses only bus macro communication. This results in a coarser grained placement grid, higher implementation overhead, and higher interface latency. Considering the reconfigurable custom instruction example that we will examine in the case study, the bus macro communication in OpenPR would introduce an overhead as high as the logic which is available in the area allocated for hosting reconfigurable submodules.

IV. Multicore Case Study

We created a reconfigurable multicore case study for demonstrating the design flow for implementing hierarchical partial reconfiguration (see Section II). The static system has been taken unchanged from a reference design which is shipped with GOAHEAD for the Atlys Spartan-6 board from Digilent. That system provides a large reconfigurable area which is divided into $15 \times 5 = 75$ tiles. Each tile is two switch boxes wide and one clock region in height and multiple adjacent tiles can be used for implementing larger modules. In addition, each tile provides an interface to a video stream that is routed meander-like over the whole reconfigurable region. The video stream follows the VGA standard but is additionally used to send control data to partial modules in the blank periods of the video stream. We can describe the system by its configuration levels:

- *level-0* The static system provides a control CPU, memory controllers for DDR and flash, video I/O modules, and an ICAP configuration controller.
- *level-1* The reconfigurable module library consists of non-hierarchically reconfigurable video processing modules (Sobel, Pong, Segmentation) and of three MIPS CPU systems which each can host up to two level-2 modules. These systems are identical except for the data memory and instruction memory layout which was selected to be 2KB:10KB, 6KB:6KB, and 10KB:2KB for instruction:data memory. This allows the execution of programs with different memory requirements by using partial reconfiguration. Each reconfigurable system provides a small text output window and can receive process data and program code via the video stream (during video idle times). A screenshot of the implemented system is shown in Figure 6.
- *level-2* The submodule library contains reconfigurable instruction set extensions (count-1-bits, mask&permute, 64-bit-parity, 4x8-bit-saturation-ADD, and CRC).

A. Implementation and Bitstream Generation

The implementation of the reconfigurable level-1 MIPS-CPU systems and the level-2 custom instructions follows exactly the design-flow that was presented in Section II. As the layout of logic, memory, and multiplier resources differs between the left and right half of the used Spartan-6 LX-45 FPGA, two physical implementations have been generated for each of the three MIPS-CPU system netlists. With the help of the design alternatives for the left and right half of the reconfigurable region, there exist 10 arbitrary placement positions for the three different CPU-systems.

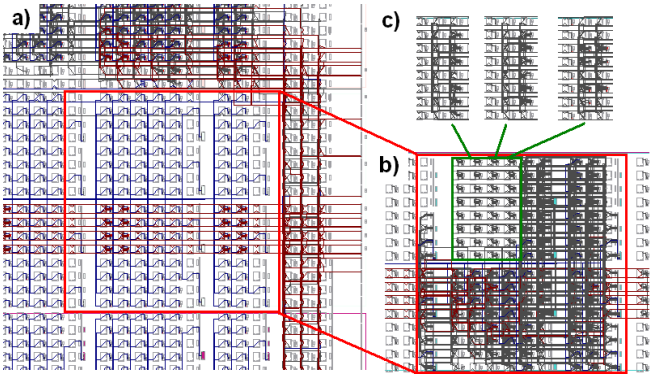

Figure 6. Hierarchically reconfigurable system. a) static system (top right corner of the reconfigurable region) b) a partial MIPS-CPU system with an attached video frame buffer overlay module to be placed into a). c) partially reconfigurable custom instructions that fit directly into the MIPS SoC.

# configurations	single level	hierarchical
1 level-1 config.	$3 \times 2 \times 4 \times \binom{4}{2} + 3 \times 2 = 150$	$3 \times 2 = 6$
1 level-2 config.	–	$2 \times 2 \times 4 + 2 = 18$
# total	150	24
bitstream storage	single level	hierarchical
1 level-1 config.	81 KB	81 KB
1 level-2 config.	–	7.8 KB / 18.6 KB
all level-1 config.	150×81 KB $= 12.2$ MB	6×81 KB $= 490$ KB
all level-2 config.	–	16×7.8 KB $+ 2 \times 18.6$ KB $= 163$ KB
storage total	12.2 MB	652 KB
configuration time	single level	hierarchical
1 level-1 config.	2.5 ms	2.5 ms
1 level-2 config.	–	240 μs / 475 μs

Table I
COMPARISON BETWEEN SINGLE LEVEL AND HIERARCHICAL RECONFIGURATION CONSIDERING 3 CPU SOFTCORE VARIANTS AND 4 (SMALL) + 1 (LARGE) ISA EXTENSIONS.

The module bounding box for a MIPS-CPU system was chosen to be 13 CLB, 2 BRAM, and 1 DSP columns wide and one clock region in height. This corresponds to a partial bitstream size of 81 KB for a level-1 module. We implemented four small reconfigurable instruction set extensions (count-1-bits, mask&permute, 64-bit-parity, 4x8-bit-saturation-ADD) that were two CLB columns wide which corresponds to 7.8 KB bitstream size per instruction. An additional CRC computation instruction was four CLB columns wide (18.6 KB).

B. Results

Table I compares a standard single level reconfigurable system implementation with a hierarchically reconfigurable system in terms of number of configurations, storage requirements and reconfiguration time. The table lists only the differences related to the three hierarchically reconfigurable MIPS-CPU systems and the five instruction set extensions because there are no differences for the other modules.

Each CPU system can be placed to any out of 10 possible positions and it is assumed that any combination of two small or one large instruction set extension can be used at run-time. Other scenarios might result in less benefits for hierarchical reconfiguration.

In the case-study, we provide $j = 4$ small modules whereof two of them can be placed in the $k = 2$ tiles for hosting submodules. This results in $\binom{j}{k}$ permutations when not considering symmetric solutions. Therefore, in the traditional single level reconfiguration case with the 3 CPU systems able to host any 2 out of 4 small ISA extensions, we would need $3 \times 2 \times 4 \times \binom{4}{2} = 144$ bitstreams. Together with the 3×2 configurations for the additional larger instruction, this adds to 150 partial bitstream permutations, hence rendering partial reconfiguration mostly useless. However, hierarchical reconfiguration allows the same flexibility with only 6 level-1 and 18 small level-2 configurations.

The table points out that the configuration memory requirements are $18.7\times$ smaller. In addition, for reconfiguring one small (large) custom instruction, only 7.8 KB (18.6 KB) instead of 81 KB configuration data have to be sent to the FPGA. This results in a $10\times$ ($5\times$) faster reconfiguration time, when using hierarchical reconfiguration.

We compared the maximal clock frequency of a hierarchically reconfigurable MIPS system with a reconfigurable MIPS system that 1) provided one instruction and 2) that provided all 5 instruction set extensions (using a slightly larger bounding box). In the first case, we found on average a small 4% performance drop, probably due to the extra constraints. However, in the second case, the hierarchical approach was 6.8% faster. Here, the hierarchical reconfiguration approach needed a smaller multiplexer than the variant supporting all instructions in parallel. As a result, this saved a logic level on the critical path.

V. Conclusions

In this paper, we introduced hierarchical reconfiguration of FPGAs which significantly reduces design complexity, FPGA resource requirements, and bitstream storage requirements for large systems consisting of modules that can share common submodules. This was demonstrated by an $18.7\times$ storage improvement and up to a $10\times$ faster reconfiguration in a reconfigurable multicore system consisting of different reconfigurable CPUs that themselves can host reconfigurable instruction set extensions.

As the next step, more realistic applications have to be developed, such as the mentioned reconfigurable database acceleration system. Furthermore, we will extend this work for allowing dynamic constructs for hardware programming which will be implemented with the help or (hierarchical) partial reconfiguration. For example OpenCL [13] is becoming popular for programming FPGAs. However, in contrast to CPU and GPU targets, FPGAs cannot create or change dynamically threads. This will become an issue for complex systems. Similarly, hierarchical reconfiguration is very suitable to implement object oriented hardware systems [14].

Acknowledgment

The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement n 318633.

References

[1] C. Claus, R. Ahmed, F. Altenried, and W. Stechele, "Towards Rapid Dynamic Partial Reconfiguration in Video-Based Driver Assistance Systems," in *Proc. of the 6th Int. Conf. on Reconfigurable Computing: architectures, Tools and Applications (ARC)*. Springer, 2010, pp. 55–67.

[2] E. J. McDonald, "Runtime FPGA Partial Reconfiguration," in *IEEE Aerospace Conference*. IEEE, 2008, pp. 1–7.

[3] M. Feilen, M. Ihmig, C. Schwarzbauer, and W. Stechele, "An Efficient DVB-T2 Decoding Accelerator by Time-Multiplexing FPGA Resources," in *22nd Int. Conf. on Field Programmable Logic and Appl. (FPL)*, 2012, pp. 75–82.

[4] C. Dennl, D. Ziener, and J. Teich, "On-the-fly Composition of FPGA-Based SQL Query Accelerators Using a Partially Reconfigurable Module Library," *Field-Programmable Custom Computing Machines, Annual IEEE Symp.*, pp. 45–52, 2012.

[5] P. Yiannacouras, J. G. Steffan, and J. Rose, "Exploration and Customization of FPGA-Based Soft Processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 266–277, Feb. 2007.

[6] A. Wold, D. Koch, and J. Torresen, "Design Techniques for Increasing Performance and Resource Utilization of Reconfigurable Soft CPUs," in *15th IEEE Symp. on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, 2012, pp. 50–55.

[7] C. Beckhoff, D. Koch, and J. Torresen, "GoAhead: A Partial Reconfiguration Framework," in *20th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2012, pp. 37–44.

[8] D. Koch, *Partial Reconfiguration on FPGAs – Architectures, Tools and Applications*. Springer, 2003.

[9] C. Beckhoff, D. Koch, and J. Torresen, "Short-Circuits on FPGAs caused by Partial Runtime Reconfiguration," in *Proc. of the Int. Conf. on Field Programmable Logic and Applications (FPL)*, Milan, Italy, Aug. 2010, pp. 596–601.

[10] C. Beckhoff, D. Koch, and J. Torresen, "Automatic Floorplanning and Interface Synthesis of Island Style Reconfigurable Systems with GoAhead," in *26th Int. Conf. on Architecture of Computing Systems (ARCS)*, 2012, pp. 303–316.

[11] Xilinx Inc., "Partial Reconfiguration User Guide (Rel. 13.2)," 2011, available online: `www.xilinx.com/support/documentation/sw_manuals/xilinx13_2/ug702.pdf`.

[12] A. A. Sohanghpurwala, P. Athanas, T. Frangieh, and A. Wood, "OpenPR: An Open-Source Partial-Reconfiguration Toolkit for Xilinx FPGAs," in *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW)*, 2011, pp. 228–235.

[13] Altera Inc., "Implementing FPGA Design with the OpenCL Standard (White Paper WP-01173-2.0)," 2012, online: `www.altera.co.uk/literature/wp/wp-01173-opencl.pdf`.

[14] N. Abel, "Design and Implementation of an Object-Oriented Framework for Dynamic Partial Reconfiguration," Ph.D. dissertation, University of Heidelberg, Germany, 2011.